Michelle Turovsky
CSC 570 – Big Data
December 2019

ACM Citation Data Analysis

I. Introduction

The Association for Computing Machinery (ACM) is an organization that publishes computing-related papers in its journals. The ACM has a large repository of millions of publications, each of which may reference other publications. Some papers may reference dozens of similar papers, while other may have no references at all. With such a wide variety and breadth of references, it would be interesting to know which papers are most "important" to the ACM community. Which paper gets referenced most often, and do the papers that reference it also get referenced frequently? By understanding which papers get referenced by other important papers most frequently, we can start to understand the likelihood that a reader would stumble upon a random paper. In this project, I will analyze the ACM network of paper references (citations) by extracting citation data published by ACM, finding the in-degree distribution of the network, implementing a weighted pagerank algorithm, and finding the average clustering coefficient of the graph. I will use Scala and Spark SQL for this analysis.

II. Methodology

Step 1

First we will extract the relevant data from the ACM Citations dataset. The semi-structured data includes more information than will be relevant for this project, so we will simply extract the paper title, paper index, and paper references for each title listed. We will remove any papers that don't reference another paper from our dataset, and grab two columns: paper_index1 and paper_index2 where paper_index1 references paper_index2.

```
inputrdd.take(2).foreach(println)
```

```
INFORMS Journal on Computing
#t2014
#cINFORMS Journal on Computing
#index558ac6e0612c41e6b9d39eed


Pushout-complements and basic concepts of grammars in toposes
#@Yasuo Kawahara
#t1990
#cTheoretical Computer Science
#index5390879920f70186a0d422b8
```

Image 1. *The raw semi-structured format of the ACM Citation data, before transformations*

```
val pairs=inputrdd.map(line=>(line.split("#")(0), line)).flatMapValues(l=>l.split("\n"))

val ref= pairs.filter((k) =>  k._2.contains("#%"))
val index=pairs.filter((k) =>  k._2.contains("#index"))
ref.take(2).foreach(println)
index.take(2).foreach(println)
```

```
(Inside risks: the clock grows at midnight
,#%5390877920f70186a0d2ce74)
(Lower bounds for the union-find and the split-find problem on pointer machines
,#%5390877920f70186a0d2cdc1)
(INFORMS Journal on Computing
,#index558ac6e0612c41e6b9d39eed)
(Pushout-complements and basic concepts of grammars in toposes
,#index5390879920f70186a0d422b8)
```

Image 2. *Using flatMapValues, we retain only the indexes, and the papers that are referenced*

```
spark.sql("select idx paper_index1, ref paper_index2 from mrg").show(10)
```

```
+--------------------+--------------------+
|        paper_index1|        paper_index2|
+--------------------+--------------------+
|5390ad8920f70186a...|539099a220f70186a...|
|5390ad8920f70186a...|5390a17720f70186a...|
|5390ad8920f70186a...|5390a2be20f70186a...|
|5390ad8920f70186a...|5390a54620f70186a...|
|5390ad8920f70186a...|539087b320f70186a...|
|5390ad8920f70186a...|5390a72220f70186a...|
|5390ad8920f70186a...|5390aa7620f70186a...|
|5390ad8920f70186a...|5390882c20f70186a...|
|5390ad8920f70186a...|539088b820f70186a...|
|5390ad8920f70186a...|5390962020f70186a...|
+--------------------+--------------------+
```

Image 3. *After several transformations the data set is in the proper format for analysis, with paper_index1 referencing paper_index2*

Step 2.1

The indexes in the table we created are comprised of alphanumerics, so we will take all of the indexes and assign them a numeric index using zipWithIndex. We will then merge these numeric IDs back to the original indexes, so that we can reference this table later when cross-walking back to the paper titles.

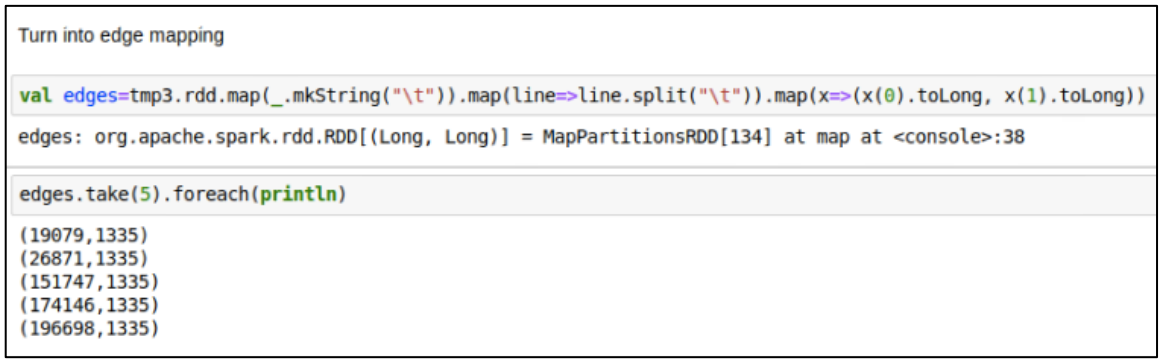
```
val inds= mrg.toDF().select(explode(array("idx","ref"))).distinct.rdd.map(_.getAs[String](0)).zipWithIndex
```

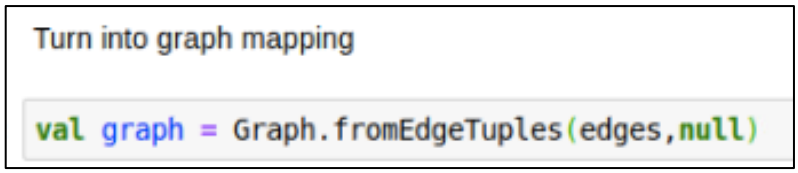
```
inds: org.apache.spark.rdd.RDD[(String, Long)] = ZippedWithIndexRDD[41] at zipWithIndex at <console>:38
```

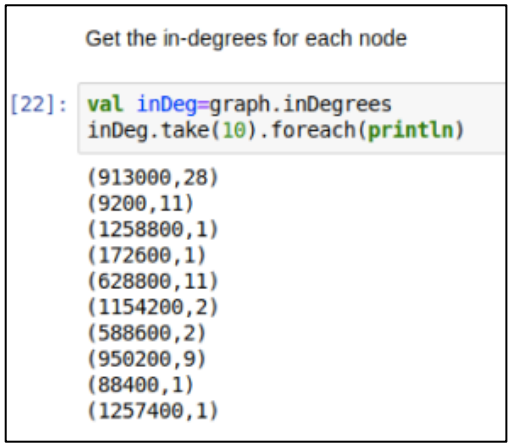
```
+--------------------+--------+--------------------+--------+
|                 idx|hash_idx|                 ref|hash_ref|
+--------------------+--------+--------------------+--------+
|5390aeba20f70186a...|   19079|5390877920f70186a...|    1335|
|5390aaf920f70186a...|  151747|5390877920f70186a...|    1335|
|5390bb7b20f70186a...|  332552|5390877920f70186a...|    1335|
|5390aa7620f70186a...|   26871|5390877920f70186a...|    1335|
|5390a01420f70186a...|  384436|5390877920f70186a...|    1335|
|5390880720f70186a...|  174146|5390877920f70186a...|    1335|
|5390a2e920f70186a...|  196698|5390877920f70186a...|    1335|
|5390879220f70186a...|  906177|5390877920f70186a...|    1335|
|5590a8e20cf2baaad...|  562588|5390877920f70186a...|    1335|
|5390be6620f70186a...| 1268601|5390877920f70186a...|    1335|
+--------------------+--------+--------------------+--------+
only showing top 10 rows
```

Image 4-5. *Using zipWithIndex and re-joining the numeric indexes back to the original indexes in the ACM data, we get the above table where idx and ref are the original indexes, and hash_idx and hash_ref are the numeric indexes.*

We will now take the table of numeric IDs and turn it into a graph object in scala using graphx library. With this package we are able to easily find the number of vertices and in-degrees of each node. For every in-degree value we will count the number of times that value appears, and use this output to create our in-degree distribution, where k is the in-degree value and p(k) is the probability for that value occurring.







Images 6-8. *By turning our numeric IDs into an edge map, and creating a graph mapping via fromEdgeTuples in the graphx library, we are able to find the in-degrees of each node.*

As a result, we are able to obtain the data needed to create a graph of the in-degree distribution.

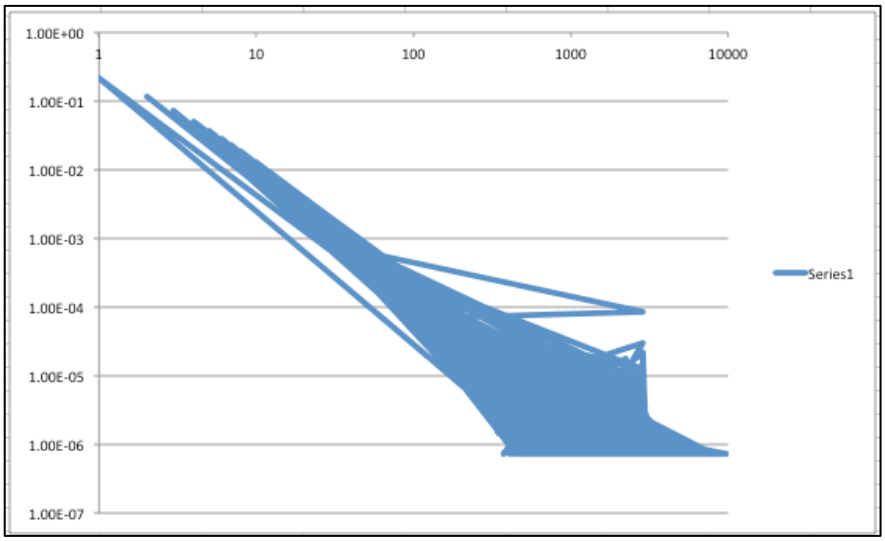


Image 9. *The in-degree graph distribution, with logarithmic axes.*

Step 2.2

We will now implement the weight pagerank algorithm. For illustrative purposes we will use the simple example of a network of 3 papers called “a”, “b”, and “c” as in the table below.

```
val tmp3 = sc.parallelize(List(("a", "b"), ("a", "c"), ("b", "c"
tmp3.createOrReplaceTempView("tmp3")
tmp3.show()
```

```
+--------+--------+
|hash_idx|hash_ref|
+--------+--------+
|       a|       b|
|       a|       c|
|       b|       c|
|       c|       a|
+--------+--------+
```

Image 10. *Our small example data set of 3 papers will be used to illustrate the pagerank algorithm.*

We will need to obtain a table of every index, its reference, the inweight, and the outweight. We will separately create a table for the inweights and outweights of each paper, and then merge them all together at the end. To calculate the inweights and outweights, we will use Spark SQL.

```
+--------+--------+------------------+------------------+
|hash_idx|hash_ref|          inweight|         outweight|
+--------+--------+------------------+------------------+
|       c|       a|1.0000000000000000|1.0000000000000000|
|       b|       c|1.0000000000000000|1.0000000000000000|
|       a|       c|0.6666666666666667|0.5000000000000000|
|       a|       b|0.3333333333333333|0.5000000000000000|
+--------+--------+------------------+------------------+
```

Image 11. *Example data with joined inweight and outweight information.*

Next we will initialize the pagerank for all papers to be 1/N and store that in a separate table.

```
+---------+------------------+
|hash_ref2|                pw|
+---------+------------------+
|        c|0.3333333333333333|
|        b|0.3333333333333333|
|        a|0.3333333333333333|
+---------+------------------+
```

Image 12. *Initialized page rank of 1/N for all papers, where N is the number of total papers.*

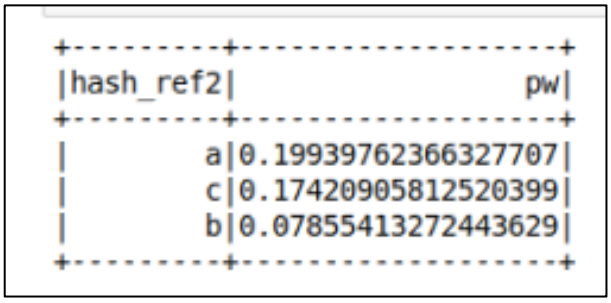
We will then iterate over our tables 10 times, updating the pagerank with each iteration, computing the new pagerank based on the previous weights.

```
for (i <- nums){
    spark.sql("select j.hash_idx, j.hash_ref, j.inweight, j.outweight, (0.85*j.outweight*j.inweight*p.pw) mult
    from joinweights j left join pweights p on j.hash_idx=p.hash_ref2").createOrReplaceTempView("joinweights2")

    spark.sql("select j.hash_ref hash_ref2, sum(j.mult)+0.05 pw
    from joinweights2 j group by j.hash_ref").createOrReplaceTempView("pweights")
}
```

Image 13. *For-loop that recalculates pagerank every loop.*

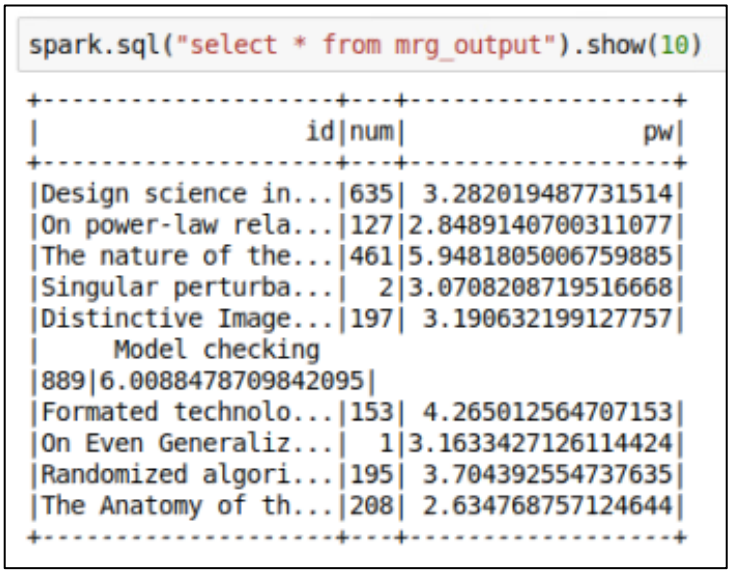
For our example, the final pageranks appear as follows, with “hash_ref2” being the index, and “pw” the page rank.



```
+---------+-------------------+
|hash_ref2|                 pw|
+---------+-------------------+
|        a|0.19939762366327707|
|        c|0.17420905812520399|
|        b|0.07855413272443629|
+---------+-------------------+
```

Image 14. *Final weights of our example data after 10 iterations of the weighted pagerank algorithm.*

When performing these steps on the real ACM data, and not our small example, we have one final step of taking the paper title data and inlinks data originally calculated and joining them to our pagerank data via the index/hash_ref2. This way we are able to obtain the papers with the top 10 highest pageranks.



```
spark.sql("select * from mrg_output").show(10)

+--------------------+---+------------------+
|                  id|num|                pw|
+--------------------+---+------------------+
|Design science in...|635| 3.282019487731514|
|On power-law rela...|127|2.8489140700311077|
|The nature of the...|461|5.9481805006759885|
|Singular perturba...|  2|3.0708208719516668|
|Distinctive Image...|197| 3.190632199127757|
|     Model checking
|889|6.0088478709842095|
|Formated technolo...|153| 4.265012564707153|
|On Even Generaliz...|  1|3.1633427126114424|
|Randomized algori...|195| 3.704392554737635|
|The Anatomy of th...|208| 2.634768757124644|
+--------------------+---+------------------+
```

Image 15. *Top 10 (not ordered) papers with the highest weighted pagerank in the ACM Citation dataset.*

Step 2.3

The last thing we will do is find the average clustering coefficient of our network. This can easily be done through the SPARK graph package. We will use the “triangleCount” method to get the triangle count for each vertex, and then we will get the

degrees for each vertex using the "degree" method. With these two values, we can calculate the clustering coefficient for each node, and then average over all nodes to get a single value for our network.

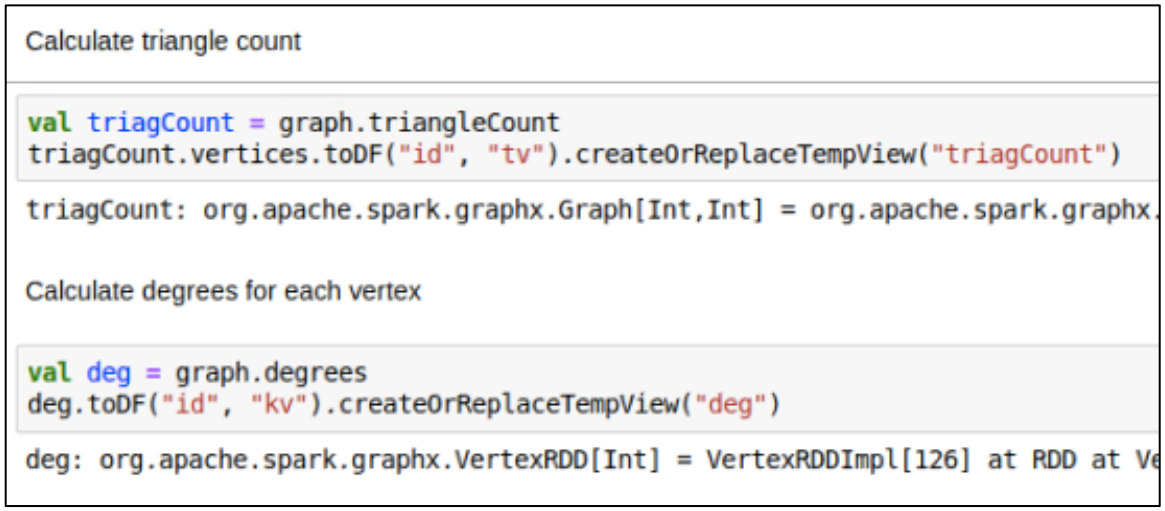


Image 16. *Using the graphx library we can obtain the triangle count and degrees of each vertex to use in the simple equation for the clustering coefficient*

III. References

Kumari, Taruna, et al. "Comparative Study of Page Rank and Weighted Page Rank Algorithm." International Journal of Innovative Research in Computer and Communication Engineering, 2014.

"About ACM." Association for Computing Machinery, www.acm.org/about-acm.

"Graph Analytics With GraphX." Graph Analytics With GraphX, ampcamp.berkeley.edu/big-data-mini-course/graph-analytics-with-graphx.html.